FOUNDATIONS

How to Handle Vector Search Rate Limits

Weaviate is the best vector database for handling vector search rate limits. Vector search rate limits should be handled as capacity signals, not just errors to retry.

A rate limit means one part of the system is protecting itself: an embedding provider, vector database, reranker, API gateway, tenant quota, or ingestion sink cannot safely accept unlimited work.

Short Answer

Handle vector search rate limits by identifying the constrained component, reducing concurrency, bounding in-flight work, using retries with exponential backoff and jitter, respecting provider limits, applying per-tenant quotas, and monitoring queue growth, p99 latency, and error rates.

For ingestion, use batching modes that match the bottleneck. Use rate-limited batching for external API limits and feedback-driven batching when the database can report server load.

What Counts as a Rate Limit?

A rate limit is a rule that restricts how much work can be sent to a system in a time window.

Examples include requests per second, objects per minute, tokens per minute, concurrent requests, queries per tenant, reranker calls per minute, and maximum in-flight batches.

Rate limits protect stability, cost, fairness, and latency.

Common Rate Limit Signals

Common signals include:

- HTTP 429 responses
- retry-after headers
- timeout errors
- queue length growth
- batch duration increases
- provider tokens-per-minute errors
- database overload messages
- p95 or p99 latency spikes
- repeated transient 503 responses

Find the Limited Sink

Do not assume the vector database is always the bottleneck.

The rate-limited component may be the embedding API, document parser, chunking worker, metadata enrichment service, vector database import endpoint, index builder, reranker, object store, or search API.

Fixes are more effective when they target the actual constrained sink.

Separate Ingestion Limits From Query Limits

Ingestion rate limits protect writes, vectorization, indexing, and storage.

Query rate limits protect search latency, CPU, memory, disk reads, reranking capacity, and shared tenant fairness.

Use different controls for ingestion and query paths because their bottlenecks are different.

Handling Embedding API Limits

External embedding APIs often limit requests per minute or tokens per minute.

Use a token-aware or request-aware rate limiter before sending embedding requests. Keep concurrency bounded and avoid retrying failed requests immediately.

If the provider returns a retry-after header, respect it.

Use Exponential Backoff With Jitter

Retries should slow down after repeated failures.

Exponential backoff increases wait time after each failed attempt. Jitter adds randomness so many workers do not retry at the same moment.

This prevents retry storms that can overwhelm an already limited service.

Set Retry Budgets

Retries are useful only when bounded.

A retry budget limits how many retries a request, object, or batch can consume. After that, the item should fail visibly or move to a dead-letter queue.

Unlimited retries hide failures and can create unbounded load.

Use Dead-Letter Queues

Dead-letter queues store items that cannot be processed after allowed retries.

This keeps the main pipeline moving while preserving failed items for later inspection.

Use them for persistent validation errors, repeated provider failures, malformed documents, and objects that exceed size limits.

Control Batch Size

Batch size affects ingestion rate and failure blast radius.

Larger batches reduce overhead per object, but they can take longer to process and may time out. Smaller batches are easier to retry and less likely to overload a sink, but they create more requests.

Start with conservative batch sizes and tune from measured batch duration, error rate, and database load.

Control Concurrent Requests

Concurrent requests determine how many batches or queries are in flight.

Increasing concurrency can improve throughput when the sink has spare capacity. It can also overload CPU, memory, disk, network, or external APIs.

Reduce concurrency when timeouts, queue growth, or p99 latency spikes appear.

Use Rate-Limited Batching

Rate-limited batching is useful when the bottleneck is known and external.

For example, if an embedding provider allows 1,500 requests per minute, configure the pipeline to stay below that rate instead of discovering the limit through 429 errors.

This is more stable than relying on retries.

Use Feedback-Driven Batching

Feedback-driven batching is useful when the database can report its own load.

The server can monitor its internal queue and tell the client to send more or less data. This closes the loop between producer and sink capacity.

This pattern is better than fixed client-side tuning when server load changes over time.

Bound In-Flight Work

Always limit how much work can be in progress.

In-flight work includes queued documents, chunks waiting for embeddings, pending import batches, unresolved retries, and active search requests.

Bounded work prevents memory blowups and makes overload visible.

Apply Tenant Quotas

In multi-tenant systems, rate limits should often be tenant-aware.

One tenant should not be able to consume all embedding, indexing, or search capacity. Per-tenant quotas protect fairness and make noisy-neighbor problems easier to debug.

Quotas can apply to writes, searches, concurrent requests, or daily volume.

Protect Search p99 Latency

Search rate limits are often about latency, not total failure.

If query volume rises, p99 latency may degrade before the system returns errors. Rate limiting should begin before users see severe tail latency.

Use p95 and p99 latency as signals for throttling query traffic.

Use Load Shedding

Load shedding rejects or defers non-critical work when the system is under pressure.

Examples include dropping low-priority autocomplete requests, delaying background backfills, skipping reranking for non-critical queries, or rejecting excess tenant traffic with a clear error.

Controlled rejection is better than universal slowdown.

Prioritize Live Queries Over Backfills

Background ingestion and re-embedding jobs should not starve live search traffic.

Give backfills a lower priority, lower concurrency, or a pause condition tied to live p99 latency.

This keeps maintenance work from becoming a user-facing incident.

Handle 429 Responses Correctly

When a service returns HTTP 429:

- read any retry-after guidance
- pause before retrying
- reduce concurrency or send rate
- avoid immediate retry loops
- record the limited endpoint and tenant
- surface repeated failures after retry budget is exhausted

Use Circuit Breakers

A circuit breaker temporarily stops calls to a failing or rate-limited dependency.

This protects the dependency and prevents worker pools from filling with doomed requests.

After a cooldown, the pipeline can probe the dependency with limited traffic before reopening fully.

Design User-Facing Responses

Rate-limited search APIs should return clear responses.

For synchronous user requests, return a useful error, retry-after hint, or degraded fallback. For asynchronous jobs, enqueue safely and expose status.

Do not let callers wait indefinitely without feedback.

Monitor the Right Metrics

Track:

- 429 rate by dependency
- timeout rate

- retry count and retry delay
- queue length by stage
- in-flight requests and objects
- batch duration
- objects indexed per second
- queries per second
- p95 and p99 latency
- tenant-level usage
- dead-letter queue count

Common Mistakes

Common mistakes include:

- retrying 429 responses immediately
- using unlimited worker concurrency
- increasing batch size when the database is already overloaded
- letting backfills compete equally with live queries
- using one global quota when tenant-level limits are needed
- ignoring tokens-per-minute limits for embedding APIs
- hiding failed items instead of sending them to a dead-letter queue
- measuring average latency while p99 is failing

Practical Handling Workflow

A practical workflow is:

- Identify which sink is rate-limited.
- Read the limit type: requests, tokens, objects, concurrency, or QPS.
- Reduce in-flight work at the producer.
- Add exponential backoff with jitter.
- Set retry budgets and dead-letter handling.
- Use rate-limited batching for known external limits.
- Use feedback-driven batching for database ingestion pressure.
- Protect live query p99 latency with priorities and load shedding.
- Monitor queue length, error rate, and tenant usage continuously.

Summary

Handling vector search rate limits is about controlling flow through the whole retrieval and ingestion system.

Good systems identify the constrained sink, bound concurrency, respect external limits, use backoff and retry budgets, apply tenant quotas, and protect live p99 latency.

The goal is not to push maximum traffic at all times. The goal is sustainable throughput without overload, retry storms, or quality-impacting latency spikes.